A User's Guide to the OPAL Compilation System (Version 2)

The OPAL Group
(edited by Christian Maeder and Wolfgang Grieskamp)

August 29, 1995

Contents

1 Introduction

For you to compile OPAL programs the OPAL compilation system needs to be installed at your site. So first of all you need to know in which directory it resides. At our site cs.tu-berlin.de this is `/usr/ocs`. In the following we will refer to this place as *ocs*. In the current Version 2.1 this directory *ocs* contains at least a file `VERSION` and subdirectories `bin` and `lib`. Generally, further subdirectories `man`, `doc` and `examples` are included.

```
> ls -1 ocs
VERSION
bin/
doc/
examples/
lib/
man/
```

Compilation of OPAL programs is controlled and driven by a single command, `ocs` or `ors`[1]. This command is located in the `bin` subdirectory of the OPAL compilation system which must be included in your search path. In our case (`csh`) this can be done by:

```
> set path=(/usr/ocs/bin $path)
```

With a `bash` the following should work:

```
> PATH=/usr/ocs/bin:$PATH
```

Generally, you will include the above or a similar line in the `.cshrc` or `.bashrc` file of your home directory, but any other means to extend your search path by *ocs*`/bin` will suffice. (Depending on your UNIX shell, you may also edit a file like `.profile, .login, .applications` or other.)

All actions regarding the OPAL compilation system are invoked by an appropriate call of `ocs`. To check if your environment is okay, execute "`ocs info`"!

```
> ocs info
You are using 'ocs-2.1'
located at '/usr/ocs'.
The project ($OCSPROJECT) is not specified.
```

If you get something different (except a more recent version number), then check your environment variables `OCS` and `OCSPROJECT`. Especially the environment variable `OCS` may refer to an old version of the OPAL compilation system, which does not support the described features. The variable `OCSPROJECT` may correctly refer to a file, usually called `ProjectDefs`, that allows inclusion of additional libraries or supports other features.

[1] The OPAL release system `ors` is based on the `shape` toolkit which extends the functionality of `ocs` by features like version control and configuration management. It accepts the same command line parameters as `ocs`. All aspects of `ors` which require some understanding of the `shape` toolkit will not be explained in this paper.

1.1 Compiling

The behavior of the ocs command is triggered by targets. In the above case the target was info. Other simple targets are help, xhelp or sendbug. Each target can be further specified via options. The default target, which can be omitted, is all, which does the compilation and linking. This target and many others require OPAL structures as arguments. Usually these arguments are given via initial "-top <Struct> <command>" or "-sub <name> <Struct>{,<Struct>} " ocs options. The latter option, which is used for subsystems, will be described in more depth in 3. In most cases, as long as you keep all your OPAL structures in one directory, ocs with -top arguments will work splendidly. In order to compile a small program like HelloWorld, you simply execute one of the following equivalent command lines in the same directory where your files HelloWorld.sign and HelloWorld.impl reside. (The program HelloWorld and other examples can be found in *ocs*/examples/.)

```
> ocs -top HelloWorld hello
> ocs -top HelloWorld hello all
```

The argument hello refers to the top-level command within the signature HelloWorld.sign. This top-level command must be a constant, *not overloaded*, *non-recursive*[2] OPAL function of type com[void][3]. The output of the above ocs command the first time you call it should look as follows:

```
gmake[1]: fopen: OCS/OcsDefs-SysDefs: No such file or directory
Generating rules for hello'HelloWorld ...
Checking Signature of HelloWorld ...
 syntax checking ...
 context checking ...
Compiling Implementation of HelloWorld ...
 syntax checking ...
 context checking ...
 translating to applicative code ...
 translating to imperative code ...
 translating to C ...
Generating sun4 object code for HelloWorld ...
Generating startup code for hello ...
Linking hello ...
```

If no errors have occurred, you may now execute the program hello.

```
> ./hello
Hello World
```

[2]Constants cannot be recursive in Opal.

[3]The type of a top-level command cannot be checked by ocs!

The line not preceded by the prompt is the output of `hello`, as implemented in `HelloWorld.impl`. If you invoke the same `ocs` command as before, you will get the following output.

```
> ocs -top HelloWorld hello
  gmake[1]: Nothing to be done for '_all'.
```

This last message indicates that there is no need to recompile or re-link anything, since nothing has changed. Usually `ocs` tries to do as little as possible in order to *make a target*.

1.2 Spelling Hints

If you misspell the name of your top-level command (below: `Hello` with a capital `H` instead of `hello`), you might get a message of this kind:

```
> ocs -top HelloWorld Hello
Generating startup code for Hello ...
.../_ostart.c: In function 'main':
.../_ostart.c:57: '__AHelloWorld_AHello' undeclared (...)
.../_ostart.c:57: (Each undeclared identifier is reported only once
.../_ostart.c:57: for each function it appears in.)
gmake[1]: *** [OCS/_HelloWorld_Hello.o] Error 1
gmake: *** [all] Error 1
```

This will also create a wrong file `SysDefs.HelloWorld-Hello` instead of `SysDefs.HelloWorld-hello`, which is of no further use and may be deleted.

If you misspell names of imported structures within your OPAL sources, you will get a message of the following kind:

```
...
  Cannot locate structure '<Name>'
...
```

In such a case make sure that `<Name>` is a legal OPAL structure that is either in the library or in the current directory. Correct possible spelling errors in your program and rerun `ocs` as before without any or the `all` target.

1.3 Targets and Options

Another important target is the `clean` target. It deletes all previously generated object files residing in an `OCS/` subdirectory of your current directory. This is useful if, for example, generated object files have been corrupted, a new incompatible compiler version has been installed or you simply want to save disk space. The next call to `ocs` with the `all` target (or no target) will recompile your

whole program. Cleaning up can also be achieved by deleting the whole `OCS/` subdirectory.

In conjunction with global optimization (see 4) full re-compilation may be necessary. Also if you have several subsystems (see 3) compiled on different computer architectures, the object code will be incompatible.

```
> ocs -top HelloWorld Hello clean
Cleaning up ...
```

If you make an error in calling `ocs`, you will get a usage message.

```
> ocs -help

usage: ocs [ -top <struct> <command> | -sub <system> <struct,...> ]
           [ <option> ... ]
           [ help | ... ]
```

This message should at least indicate that `-help` is not an option but a *target* `help`, which like `info`, `xhelp` and `sendbug` does not need any Opal structures as further arguments.

An interesting *option* is `-v3` which allows observation of minor compilation steps.

```
> ocs -top HelloWorld hello -v3
```

Should the compiler crash, the verbose output could be very useful in locating the trouble (see 5). The following sections will give more details on the way `ocs` works and explain the advanced features and options of this compilation driver.

2 What Really Happens

The casual reader may skip this section. The Opal compilation system keeps all compilation products, so-called objects, as well as diagnostic files `*.diag`[4] in a subdirectory `OCS/`. A special object is an internal `OCS/OcsDefs-SysDefs` [5] file holding the dependencies imposed by import relations of all source and object files. This `OCS/OcsDefs-SysDefs` file is actually a *makefile*, which is automatically generated by `genmake` and passed as argument to a call of GNU `make` or `gmake`. GNU `make` is triggered by a `SysDefs` file which resides in your source directory and is a direct translation of your `ocs` command line. The `SysDefs` file for the above example looks as follows:

[4]When editing source files with an emacs you may be interested in the opal browser (*ocs*`/lib/emacs/opal-mode.el`) to keep track of syntax and type errors, but also to obtain detailed information of successfully compiled sources.

[5]When using `ors` and `shape` the dependency file is called `OCS/OrsDefs-SysDefs`

```
TOPSTRUCT=Hello
TOPCOM=hello
GENOPTIONS=-V -v3
GENSUBSYS= $(GENSTDSYS)
SOURCES=$(SIGNS) $(IMPLS)
COMPONENTS=$(SOURCES)
include $(OMLIBPATH)/GlobalRules.top
```

If `SysDefs.HelloWorld-hello` is renamed or linked to `SysDefs`, you may invoke compilation by `ocs` without any further arguments. This is particularly useful if your command line has become rather long, i.e. in conjunction with subsystems or additional options as described in 3. You may even edit the file `SysDefs` yourself to change the behavior of `ocs` without having to assemble a complete new command line. If you have established a link to `SysDefs`, you may alternatively call `ocs` without arguments to rerun `ocs` as before or with different arguments, which will update your `SysDefs` file. (However, this does not work if you change your top-level command or top-level structure, because then a different file is created.) In general, within bigger projects, it may be much more convenient to work only with such `SysDefs` files.

Usually, `gmake` requires further *rules* to build a target. These are kept in the OPAL maintenance subdirectory *ocs*`/lib/om`. The actual programs to analyze the import relation and compile OPAL structures are *ocs*`/bin/opalimports`, *ocs*`/bin/genmake` and the front-end *ocs*`/bin/oc1` in conjunction with the back-end *ocs*`/bin/oc2`. Both *ends* are also called `oc` for OPAL compiler.

The program `opalimports` extracts names of imported structures from sources and writes them to `*.deps` files, which are used by `genmake` to create the file `OcsDefs-SysDefs`. The front-end `oc1` generates export files (`*.exp`) from *signatures* and analyzed[6] OPAL (`*.ana`) from *implementations*. The results of the back-end `oc2` are `*.opt` files to allow global optimization and C sources (`*.c` and `*.h`), which will be further compiled by a C compiler like `gcc` to object code (`*.o`). Eventually, object code will be linked together with a linker (`ld`) to an executable program like `hello`.

A further possibility to influence the behaviour of `ocs` or `gmake` is to include an additional makefile containing project definitions. This makefile, usually called `ProjectDefs` and which may be located anywhere, will be included if your environment variable `OCSPROJECT` refers to this file with its complete path. With "`ocs info`" you can check if this environment variable is set.

A Editing `ProjectDefs` files requires some knowledge about the options which may be passed to the individual programs. But in principle you just define `make` -macros or -variables as in `SysDefs` files (see 4). You never create a `ProjectDefs` file from scratch yourself but simply modify a copy of the template *ocs*`/lib/om/tmpls/ProjectDefs.tmpl`.

[6] This is actually *applicative code* that differs from *inter-Opal*

The order of inclusion of various makefiles which define *make variables* and *targets* (see *ocs*`/lib/om/make/Makefile.develop`) is as follows:

`ProjectDefs` file
ocs`/lib/om/specs/Specs.os.auto`
`SysDefs` file
ocs`/lib/om/make/GlobalRules.top`
`OCS/OcsDefs-SysDefs` file
ocs`/lib/om/make/GlobalRules`

Variable definitions which may be given on the `ocs` command line itself are considered before a `ProjectDefs` file.

3 Constructing a Private Library

Large projects are typically distributed over several directories. For this reason `ocs` (and especially `ors`) supports compilation of more or less unrelated (non-top-level) OPAL structures within one directory. Compilation in such a directory, which is then called a subsystem, is invoked via `-sub` as the first `ocs` argument.

Consecutive arguments of `-sub` are the directory name and a comma-separated list of structure names (either in quotes or without blanks).

```
> cd ../trees
> ocs -sub trees Tree,TreeMap,MkTree,TestTree
```

This command assumes that there are four structures (i.e. eight files `Tree.sign`, `Tree.impl`, `TreeMap.sign`, `TreeMap.impl`, `MkTree.sign`, `MkTree.impl`, `TestTree.sign` and `TestTree.impl`) in the current directory `trees`. In case that `TestTree` imports all the other structures (similar to a top-level structure) it would suffice to run:

```
> ocs -sub trees TestTree
```

The above command is almost identical to that for a top-level structure[7], but instead of an executable program a library archive `OCS/libtrees.a` will be created. This new library may be referred to later on by the full name of the directory `trees`.

Suppose you have established another directory `inout` to hold your own I/O functions. If all structures are independent of `trees`, an independent other library can be set up:

[7]Our naming convention is that OPAL structures start with a capital letter, whereas top-level commands such as all OPAL functions and directory names start with a lower-case letter.

```
> cd ../inout
> ocs -sub inout Read,Write
```

All so-called subsystems (as well as top-level systems) may import structures from the standard OPAL library, because these are implicitly included by `ocs`. But if structures from self-defined subsystems need to be imported elsewhere, then these subsystems must be referred to on the `ocs` command line. Suppose your I/O functions are meant to handle trees. In that case your subsystem `inout` should be compiled as follows:

```
> ocs -sub inout Read,Write -s ../trees
```

The dots correspond to the proper (relative) path to the directory `trees`. Take a look at the `SysDefs` file (`SysDefs.inout`) generated by the above command:

```
NODENAME=inout
STRUCTS=Read Write
GENOPTIONS=-v1
GENSUBSYS= -s ../trees $(GENSTDSYS)
SOURCES=$(SIGNS) $(IMPLS)
COMPONENTS=$(SOURCES)
include $(OMLIBPATH)/GlobalRules.sub
```

The `NODENAME` corresponds to the directory name of the subsystem. `STRUCTS` lists the structures of the current subsystem, `GENSUBSYS` other dependent subsystems. `GENSTDSYS` stands for the standard OPAL library.

Both subsystems may be referred to within a third directory, e.g.:

```
> cd ../main
> ocs -top Main run -s ../inout -s ../trees
```

3.1 Hierarchy of Subsystems

In a project with many subsystems and possibly several top-level systems it is useful to have a single `SysDefs` file in each directory to hold subsystem dependencies, because then a simple call of `ocs` without any parameter will do all the necessary compilation in each directory. However, all directories will still need to be compiled from the bottom up w.r.t. their dependencies, i.e. libraries to be used must be compiled first. This task can be avoided by creating a special `SysDefs` file in yet another directory. In our case we would create a `SysDefs` file like *ocs*`/lib/om/tmpls/SysDefs.misc` in the parent directory of `trees`, `inout` and `main` as follows:

```
SUBNODES=trees inout main

_default: all

_all:

_clean:

_check:

include $(OMLIBPATH)/GlobalRules
```

A SysDefs for ocs behaves like a Makefile for make, i.e. variables and targets may be defined. The special variable SUBNODES lists directories which need to be handled by ocs *prior* to the *current* directory. The order of the listed subnodes is of course significant because trees should be compiled before inout and inout before main. The *underline targets* are used for recursion control, as defined in *ocs*/lib/om/make/GlobalRules.

The above parent directory itself may be viewed as an ordinary *subnode* from yet another node, and thus a large hierarchy between directories (i.e. subnodes) may be established.

At a *top node* a call of ocs with target cleanall would remove all objects of all *subnodes* and complete recompilation could be achieved by a simple call of ocs.

If no subnodes are given, a call of ocs will only compile the sources of the current directory, assuming that subsystems given by the variable GENSUBSYS have been properly compiled previously, either explicitly or implicitly by a call from an *upper node*.

In general a *node* is simply a directory with at least a single file SysDefs. Some *nodes* may be *subsystems* that form a library of OPAL structures. The hierarchy between *subsystems* is established by the OPAL import relation and described by GENSUBSYS entries, whereas the hierarchy between *nodes* must be set up manually by defining the variable SUBNODES in SysDefs files. Both hierarchies are independent of the *subdirectory* relation, but care should be taken that a hierarchy of *subnodes* does not contradict the hierarchy of *subsystems*. It would also be quite uncommon (but possible) to place top nodes in subdirectories of subnodes.

3.2 Alternative Hierarchies

The variable SUBNODES may also be defined in the SysDefs file of the directory main as follows:

```
SUBNODES=../trees ../inout
```

```
TOPSTRUCT=Main
TOPCOM=run
GENOPTIONS=v1
GENSUBSYS=-s ../inout -s ../trees $(GENSTDSYS)
SOURCES=$(SIGNS) $(IMPLS)
COMPONENTS=$(SOURCES)
include $(OMLIBPATH)/GlobalRules.top
```

In this case `main` becomes the *top node* and the special `SysDefs` of the parent directory is no longer necessary. It would also make sense to let `main` be the parent directory of `trees` and `inout`. This would shorten the relative path names to the right of the variables `SUBNODES` and `GENSUBSYS`.

Because `inout` depends on `trees`, the file `inout/SysDefs` might just as well contain `../trees` in its `SUBNODES` line. In this case the subnode `trees` could be omitted from `main/SysDefs` because `trees` has then become an indirect subnode of `main`.

Summarizing, all alternatives described above are only recommended for small projects. Within bigger projects many subsystems rarely change and need not always be checked by the `SUBNODES` mechanism. Only the final overall compilation would be performed from a top node, as described in 3.1

4 Additional Options

The `ocs` command also passes a couple of options to the OPAL compiler `oc`. An overview is listed in "`ocs xhelp`". The verbose and warning-level options simply modify the visible output of the compiler. Level zero `-v0 -w0` enforces quiet compilation and will suppress all warnings. This level is intended for compilation batches where the output can be ignored. The default options `-v1 -w1` (or `-v -w`) will display major compilation steps and give warnings in cases of unusual or error-prone OPAL source code. With `-v3 -w2` even minor compilation steps and further hints will be displayed.

The options `-d` and `-o` influence only the object code generated by the back-end `oc2`. The default is to ignore debugging. With `-dd` or just `-d` you may generate code for use with an object code debugger. This option is also necessary for producing a back-trace of a crashing program (see 5.2) later on. The options `-dT` or `-dt` serve to trace function entries and exits of all or only exported functions respectively.

In conjunction with optimization, debugging will be very difficult, because it is difficult to associate object code with your original OPAL sources.

$\mathcal{E}$ It is already difficult to associate OPAL sources with corresponding C sources. You must pass the `-keep` option to `ocs` to study `*.c` files. With this option set the intermediate compilation products `*.c`, and `*.ana` files will not be removed after creation of `*.o` object files. Within a subsystem that you do not

want to change any more, you may even delete all `*.o` files because all object code is stored in an archive `OCS/lib<NODENAME>.a`

The default for optimization is to perform no optimization. Special optimization methods can be switched on individually by `-o<letters>`, where the letters may be any combination of {`e, u, s, c, m, p, g, T, S, C`}. These stand for elimination of common subexpressions, unfolding of function definitions and equations, simplification of expressions, constant evaluation only once, memory allocation optimization, partialities optimization, global inter-structure optimization, time-consuming optimizations, speculative optimizations and C compiler optimizations, respectively. Optimization flags are either passed to the back-end `oc2` or to the C compiler `gcc`.

You usually do not need to set all these options, rather only specify `-o` or `-o1` (these stand for `-oeucmC`) or `o2`, dthereby additionally setting `-opgT`, which performs time consuming global optimization.

$\boxed{\mathcal{A}}$ The settings of your `ocs` default behaviour may switch on special debug and/or optimize flags. In this case you cannot switch off these flags except by editing your `SysDefs` file variable `GENOPTIONS`.

`ocs` also supports the OPAL property language. With the `-prop` option set, dependency rules are generated for the corresponding sources `EXTERNAL` and `INTERNAL PROPERTIES` with extensions `*.extp` and `*.intp`. Context checking of these properties is invoked by the `check` target of `ocs` and will result in so called *inter-Opal* files `*.inter`. Inter-Opal is intended as a common interface to other OPAL tools, like the existing `browser`, a documentation system `DOSFOP` or the proof-checker `BOP`. If you establish a whole subsystem with property sources, then this subsystem should be included with `-sp` either on the `ocs` command line or in your `SysDefs` file to the right of `GENSUBSYS`.

$\boxed{\mathcal{A}}$ The modifiers `d` and `f` for inclusion of subsystems also control the generation of dependencies. Option `-sf` includes a *frozen* subsystem, which does not need to hold the source files, while `-sd` corresponds to the default `-s`.

4.1 Customizations

Instead of specifying options as described above, it is possible to define special variables on the command line. For example, after a long period of development and testing you would release your program fully optimized as follows.

```
> ocs -top HelloWorld hello opt=full debug=no
```

If you are using additional subsystems, you may first do an `ocs` call with the `cleanall` target to enforce recompilation with full optimization for all subsystems later on.

Another way to modify the behaviour of `ocs` is to set up a `ProjectDefs` file. With this you are able to override settings that are usually fixed by files located

in the *ocs*/lib/om/specs directory. Whenever you are using a ProjectDefs file, which must be included by setting the environment variable OCSPROJECT, you may choose an *experimental* compiler.

```
> ocs -top HelloWorld hello ocs=expocs
```

The value expocs of the variable ocs will enable a set of experimental variables within in your ProjectDefs file. The following is an incomplete list of make variables for ProjectDefs files.

CLDLIBS: list of libraries used by the linker ld
CLDLIBPATH: path for the linker to look for libraries
ocs: may be stdocs or expocs
debug: may be no, c or opal
opt: may be no, modest, medium or full
EXP_OC1: alternative front-end oc1
EXP_OC2: alternative back-end oc2
EXP_OC1FLAGS: flags for oc1
EXP_OC2FLAGS: flags for oc2
EXP_GENSTDSYS: alternative standard libraries

If one of these EXP_... variables is not defined, the standard values (i.e. STD_OC1) will be used. By using "+=" when assigning values to variables the standard values (flags) can be easily extended. If you use ":=" instead, the variable will be completely redefined. Note that variables can also be set on the command line with a simple "=" without blanks or within quotes.

The following example will include a tcl library and measure the compilation time of the front-end.

```
CINCPATH      := -I/usr/tcl/include
CLDLIBS       := -ltk -ltcl -ldpnetwork -lnsl -lXpm -lX11 -lsocket
CLDLIBPATH    := -L/usr/tcl/lib -L/usr/X11/lib
ocs := expocs
EXP_OC1 := time /usr/ocs/bin/oc1
EXP_OC1FLAGS += -ztraceIO
```

Instead of setting ocs := expocs within your ProjectDefs file, you generally achieve more flexibility by setting this variable on the command line.

5 Error Handling

If something goes wrong or behaves unexpectedly you will first have to find the source of the trouble. It may be one of the following:

- your environment, search path, `gmake`, `gcc`, other UNIX shell commands, `SysDefs`- or `ProjectDefs`-files
- the OPAL compilation system
- front-end or back-end of the OPAL compiler
- your OPAL sources

The first thing to do is to check your environment by executing "`ocs info`". Make sure that you are in the directory where your sources reside. Check your `ocs` command line for proper arguments[8]. Both `-top` and `-sub` options expect exactly two further arguments, either a top-level structure and a top-level command or a subnode's name and a list of structures as one argument without blanks or within quotes.

If you are using your own `SysDefs` file, check the variables described above. Make sure that `GENOPTIONS` contains the `-V` option in order to produce verbose output. If necessary, also add `-v3` and `-w2` options, which will be used by the compiler to achieve maximum verbosity. Error messages from the compiler can be recognized by the word `ERROR`, followed by a position given as a row and column number. The verbose output of the compiler should enable you to determine the current compiler phase and your current source part.

Compilation of C sources may fail because specific header files could not be found. This may be fixed by defining the variables `CINCPATH` or `EXP_CCFLAGS`.

If something goes wrong during linking, the reason might be a wrong top-level command name or that required libraries are not accessible. If system libraries are not accessible, ask your system administrator. Some libraries may be included by defining the variables `CLDLIBS` and `CLDLIBPATH`. Make sure that your own subsystem libraries are properly compiled and archived. Once you have an executable program the following unwanted outcomes during runtime are possible:

- `RUNTIME ERROR`
- `Segmentation Fault`
- `Bus Error`
- Non-Termination

An OPAL `RUNTIME ERROR` indicates a programming error, i.e. you have applied a partial function to a variant of an object it was not defined for. For example, you applied the function `ft'Seq` to an empty sequence or `cont'Option` to `nil`.

[8]Targets must *not* be preceded by a hyphen

A `Segmentation Fault` may have several causes. One may be that your program has used up all memory or other resources. Try to observe the memory usage with a UNIX command like `top`.

A `Bus Error` usually indicates corrupted object code. For example, such code will be produced if your top-level command was not of type `com'Com[void'Void]` or the top-level command name was overloaded. Another reason may be that objects from subsystems have been compiled on a different architecture. In this case recompile your whole program by first deleting all your objects with the `cleanall` target from `ocs`.

Apparent or actual non-termination are caused by endless recursions or rather long computations of possibly exponential order. In such a case abort program execution with `Ctrl-c` or the UNIX kill command. Provided your program was compiled with the debugging option `-d`, a back-trace generated by `btrace` (see below) may be useful to locate the bug.

5.1 Sending Bugs

Unfortunately the OPAL compiler itself may sometimes crash in one of the ways described above. This is of course a serious bug, but compiler crashes are far more likely if flawed programs are compiled. Although you will get no advice as to which problem may have crashed the compiler, there are most probably errors in your sources too. If you think your sources are correct, you are requested to consult the document *How to Identify and Workaround Bugs of the* OPAL *compiler*. If this document cannot help you, you have probable discovered a new compiler bug. We would be very grateful if you sent this bug to us using the `sendbug` target from `ocs`. But before sending a bug, please make sure that the bug is reproducible by fully recompiling all your sources after all objects have been removed (with target `cleanall`). Also make sure that you are using a reasonably new compiler version and that a possible `ProjectDefs` file does not refer to a different compiler. At least execute "`ocs info`"! Compilation should be run verbosely `-V -v3` in order to facilitate your categorization of the bug. Of course, it would be great if you can give some sort of analysis of the compiler bug, which you may have gained when trying to work around it. Even better, mail us a successful workaround!

```
> ocs sendbug

This script simplifies the report and analysis of OCS bugs.

In the following, you will be asked several questions. If you
wish to include the sources which raised the problem, you
will be asked in particular for the directory path to a
system which contains the sources.
```

```
Do you wish to continue? (y/n) y

What do You suggest the bug is related to?
Categories are:
 1 = syntax analysis
 2 = context analysis
 3 = code generation
 4 = standard library
 5 = maintenance system, environment
 6 = other
Your choice?
```

You will also be asked what your `ocs` command line looks like. When you answer this question do not include any target like `all, gen, clean, check` or `pack`. When being asked to comment on the bug it would be helpful if you could paste at least your error message into the mail buffer.

If you have more general problems, comments or questions on Opal, you might like to mail us at `opal-users@projects.uebb.tu-berlin.de`.

5.2 Debugging

The support for debugging Opal programs is very poor at the moment. Therefore, the best debugging strategy is simply checking Opal sources.

The C debugger can be used to inspect the dynamic call chain of crashed programs or to trace the execution in the Opal sources. This is realized by using the `#line` directive of the C preprocessor. In order to use this feature, you must have supplied the option `-dd` to `ocs`.

Two scripts based on the GNU debugger `gdb` are bundled with `ocs`.

1. The script `btrace`, when prefixed to a command running an Opal program, runs the program under `gdb`, analyzes its output, and given a crash produces a back-trace of the form:

   ```
   Foo.impl: 110
   Foo.impl: 138
   ...
   ```

   This script mainly filters out those elements of `gdb`'s back-trace which are practically impossible to understand for someone not familiar with the coding details.

2. The script `debug` operates in a similar manner to `btrace`, except that you end in the `gdb` command line interpreter. With `up` and `down` you can walk through the stack frame, and with `list` you can view the corresponding OPAL sources.

Another alternative is to use the `gdb-mode` which comes with emacs.

5.3 Debug Prints

The structure `DEBUG` from the standard library provides ways of spreading side-effect prints through the OPAL code. This can be used to explicitly trace certain program points, including the output of structured data objects at these points. The conversion of a particular data type into a textual representation must, however, be realized by the user.

Note that when using the functions from `DEBUG` the OPAL compiler will always eliminate dead code, even if optimization is not enabled; therefore, you have to feign a use of the side-effect functions. In the following example, the side-effect print is *not* performed:

```
DEF f(X) == LET _ == PRINT(true,"Hello, I'm f",X) IN X
```

To achieve your objective you have to write something like:

```
DEF f(X0) == LET X == PRINT(true,"Hello, I'm f",X0) IN X
```

This works because `PRINT` is identity in the third argument, and because the compiler does not know this, it cannot optimize it away.

If you simply use `writeLine'Stream` within your commands to trace program execution, be aware that the arguments of following commands may be evaluated before you have started writing:

```
writeLine(stdOut, "Hello") &
writeLine(stdOut, f(N)`)
```

In the above example the arguments of the second `writeLine` are evaluated before the first `writeLine` has been executed. Thus, if function `f` crashes, no output will appear on the screen. To avoid this strange behaviour supply a dummy argument to the second command:

```
writeLine(stdOut, "Hello") & (\\ _ .
writeLine(stdOut, f(N)`))
```

Now evaluation of the second argument of `&'ComCompose` will only occur after the first line has been written.

Of course you cannot use `writeLine` within functions that are not commands, i.e. functions of type `com'Com`. Therefore `PRINT'DEBUG` should be the preferred choice to display execution points.